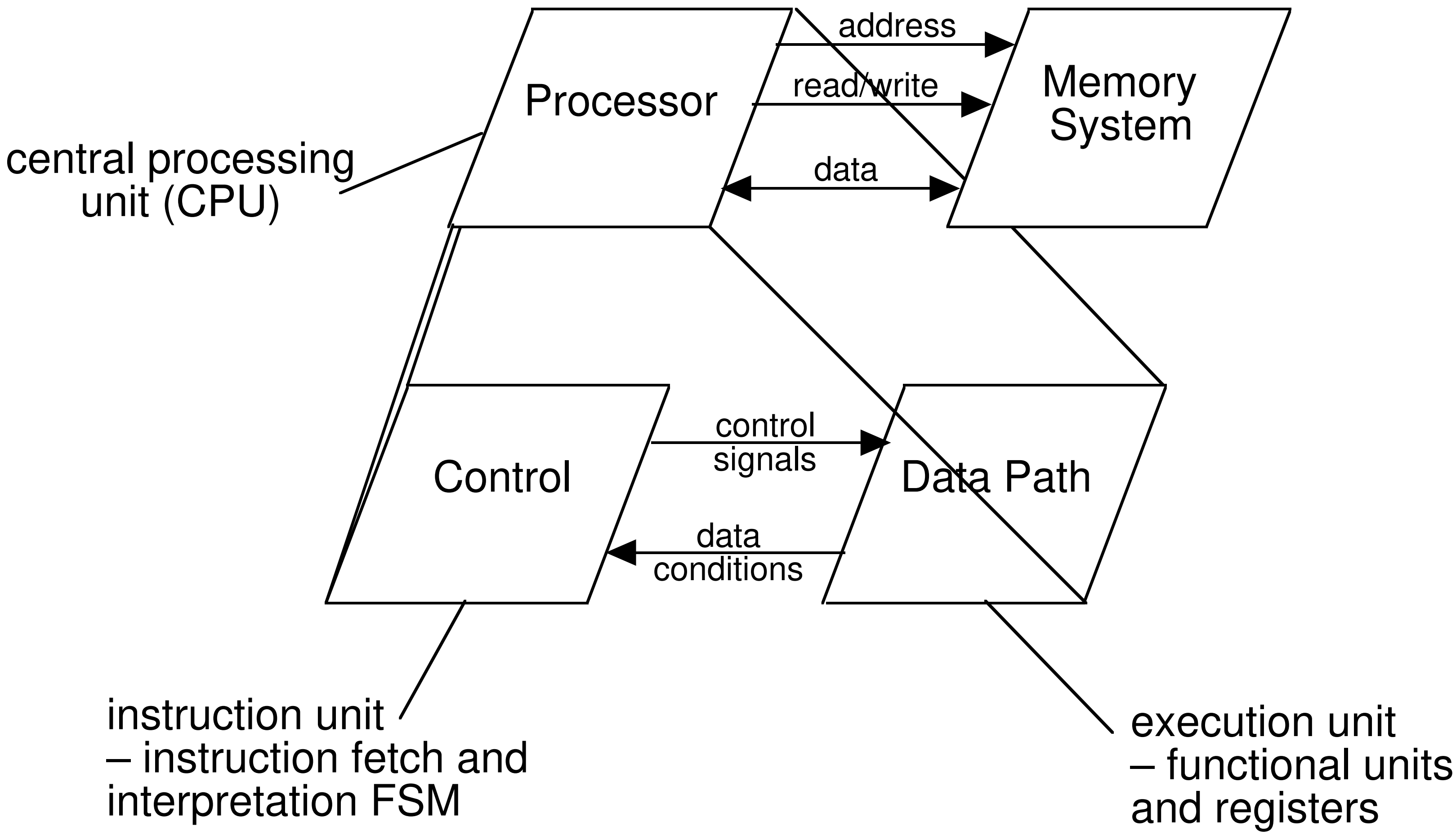
Computer organization

- **Computer design – an application of digital logic design procedures**
- **Computer = processing unit + memory system**
- **Processing unit = control + datapath**
- **Control = finite state machine**
  - inputs = machine instruction, datapath conditions
  - outputs = register transfer control signals, ALU operation codes
  - instruction interpretation = instruction fetch, decode, execute
- **Datapath = functional units + registers**
  - functional units = ALU, multipliers, dividers, etc.
  - registers = program counter, shifters, storage registers

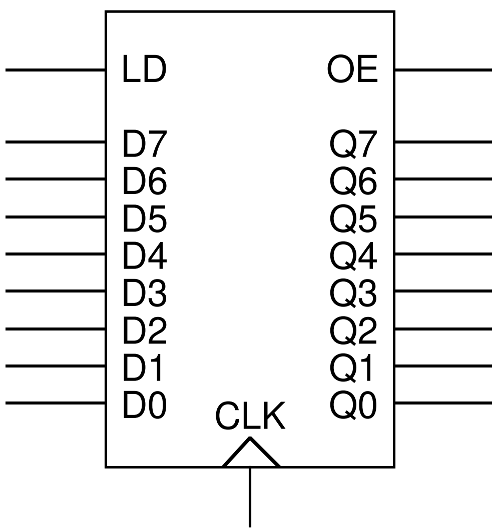
Structure of a computer

- **Block diagram view**



Registers

- **Selectively loaded – EN or LD input**
- **Output enable – OE input**
- **Multiple registers – group 4 or 8 in parallel**

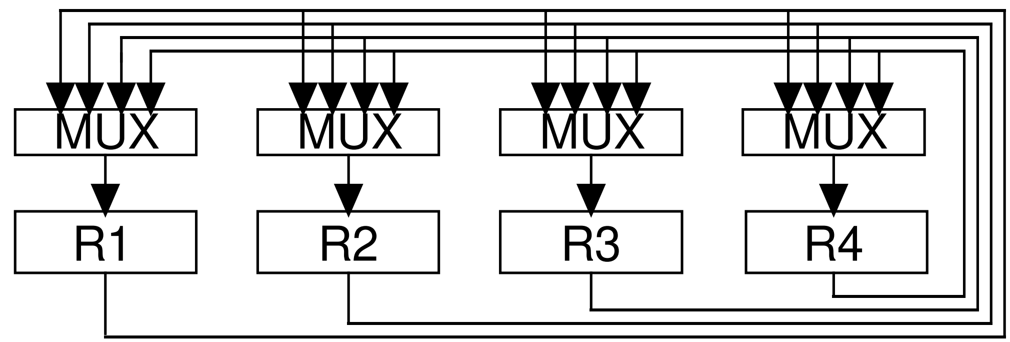


OE asserted causes FF state to be connected to output pins; otherwise they are left unconnected (high impedance)

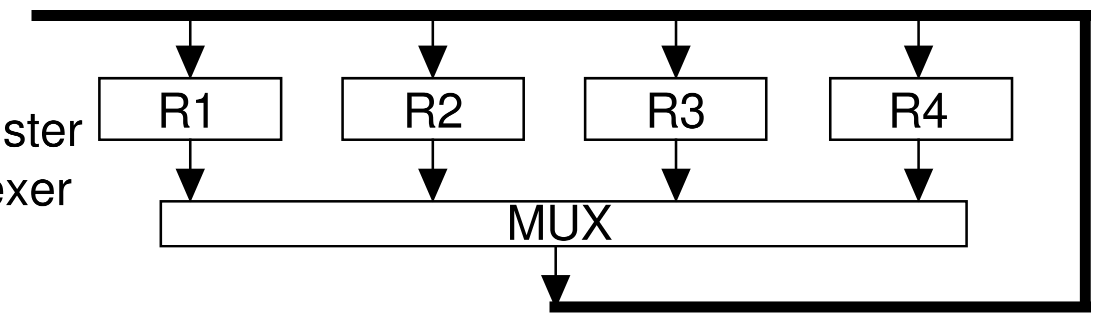
LD asserted during a lo-to-hi clock transition loads new data into FFs

Register transfer

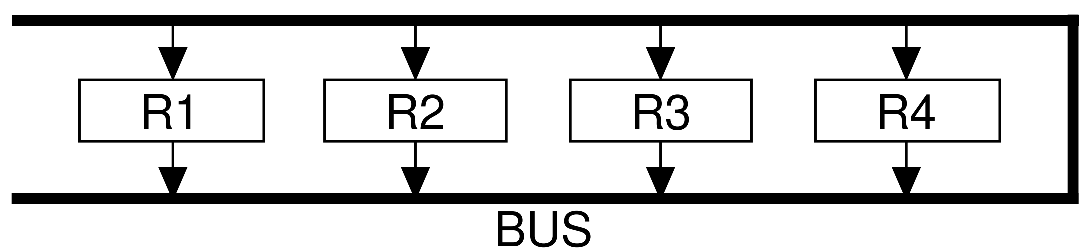
- **Point-to-point connection**
  - dedicated wires
  - muxes on inputs of each register



- **Common input with multiplexer on outputs**
  - load enables for each register
  - control signals for multiplexer



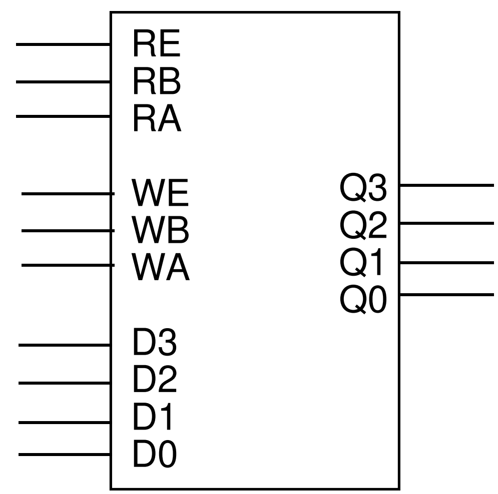
- **Common bus with output enables**
  - output enables and load enables for each register



Register files

- **Collections of registers in one package**
  - two-dimensional array of FFs
  - address used as index to a particular word
  - can have separate read and write addresses so can do both at same time

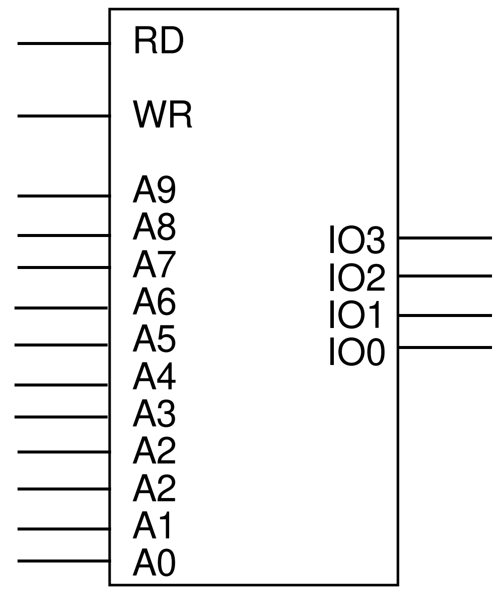
- **4 by 4 register file**
  - 16 D-FFs
  - organized as four words of four bits each
  - write-enable (load)
  - read-enable (output enable)



Memories

- **Larger collections of storage elements**
  - implemented not as FFs but as much more efficient latches
  - high-density memories use 1 to 5 switches (transitors) per memory bit

- **Static RAM – 1024 words each 4 bits wide**
  - once written, memory holds forever (not true for denser dynamic RAM)
  - address lines to select word (10 lines for 1024 words)
  - read enable (same as output enable, often called chip select)
    - can view as read enable
    - permits connection of many chips into larger array
  - write enable (same as load enable)
  - bi-directional data lines
    - output when reading, input when writing



Instruction sequencing

- **Example – an instruction to add the contents of two registers (Rx and Ry) and place result in a third register (Rz)**

- **Step 1: get the ADD instruction from memory into an instruction register**

- **Step 2: decode instruction**
  - instruction in IR has the code of an ADD instruction
  - register indices used to generate output enables for registers Rx and Ry
  - register index used to generate load signal for register Rz

- **Step 3: execute instruction**
  - enable Rx and Ry output and direct to ALU
  - setup ALU to perform ADD operation
  - direct result to Rz so that it can be loaded into register

Instruction types

- **Data manipulation**
  - add, subtract
  - increment, decrement
  - multiply
  - shift, rotate
- **Data staging**
  - load/store data to/from memory
  - register-to-register move
- **Control**
  - conditional/unconditional branches in program flow
  - subroutine call and return

Elements of the control unit (aka instruction unit)

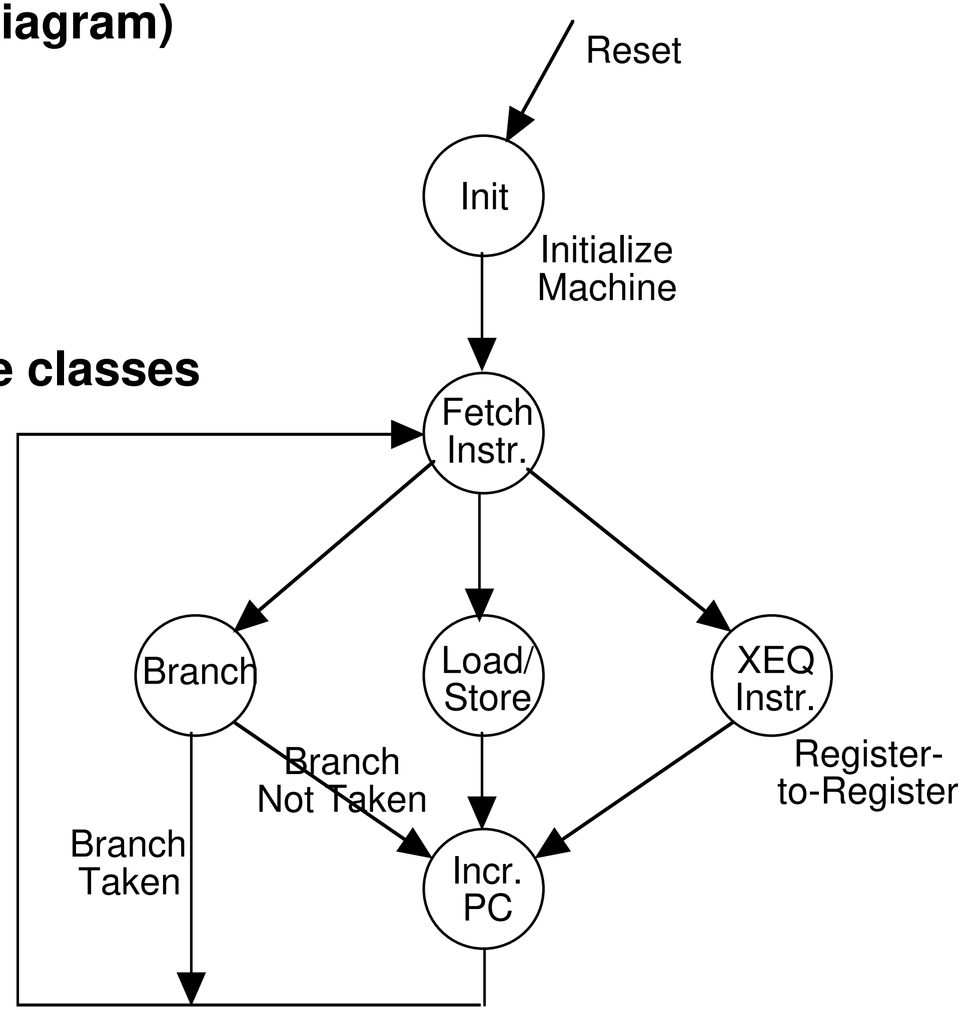
- **Standard FSM elements**
  - state register
  - next-state logic
  - output logic (datapath control signalling)
  - Moore or synchronous Mealy machine to avoid loops unbroken by FF
- **Plus additional "control" registers**
  - instruction register (IR)
  - program counter (PC)
- **Inputs/outputs**
  - outputs control elements of data path
  - inputs from data path used to alter flow of program (test if zero)

Instruction execution

- **Control state diagram (for each diagram)**
  - reset
  - fetch instruction
  - decode
  - execute

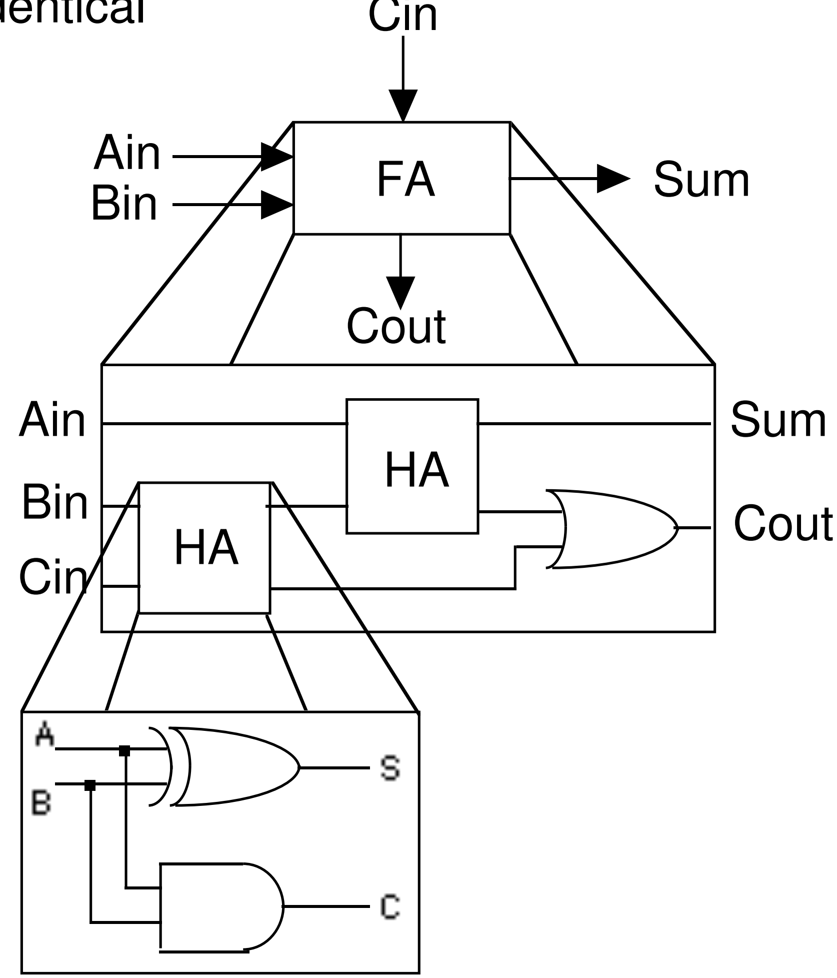
- **Instructions partitioned into three classes**
  - branch
  - load/store
  - register-to-register

- **Different sequence through diagram for each instruction type**



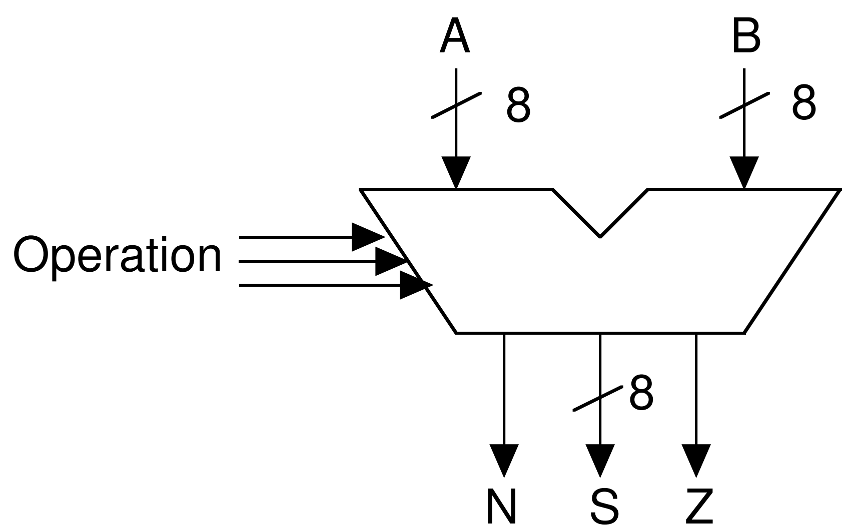
Data path (heirarchy)

- **Arithmetic circuits constructed in hierarchical and iterative fashion**
  - each bit in datapath is functionally identical
  - 4-bit, 8-bit, 16-bit, 32-bit datapaths



Data path (ALU)

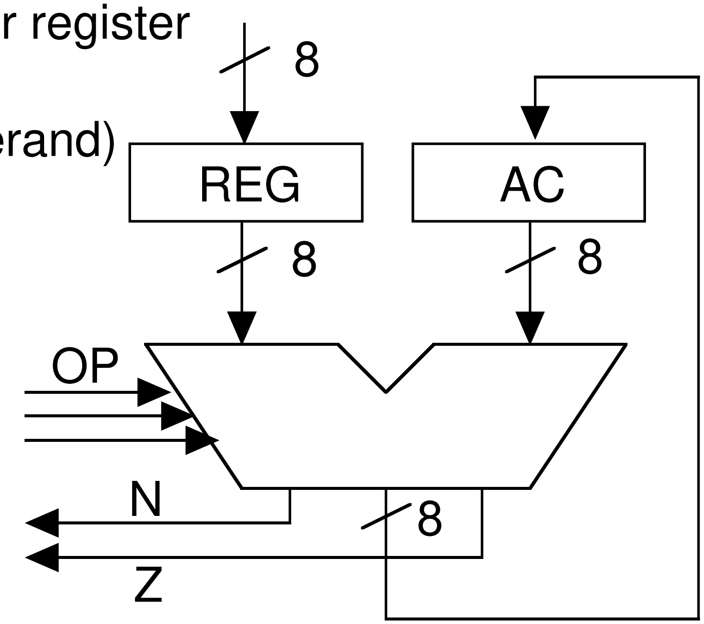
- **ALU block diagram**
  - input: data and operation to perform
  - output: result of operation and status information



Data path (accumulator architecture)

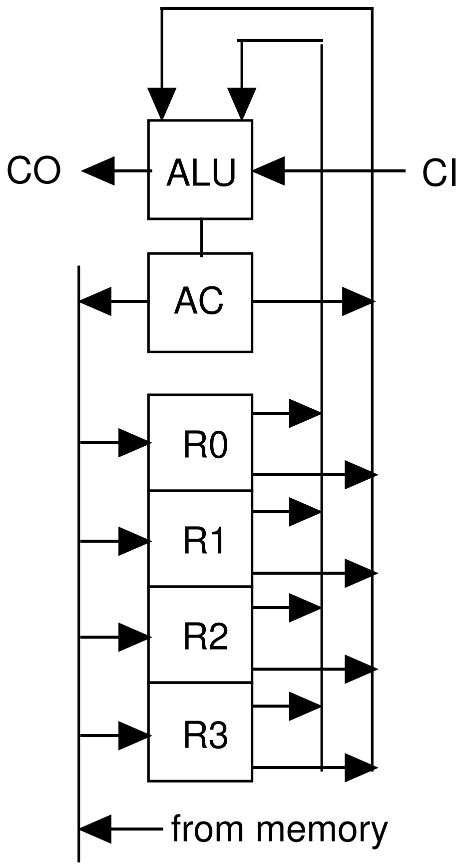
- **Accumulator**
  - special register
  - one of the inputs to ALU
  - output of ALU stored back in accumulator

- **One-address instructions**
  - operation and address of one operand
  - other operand and destination is accumulator register
  - AC <– AC op Mem[addr]
  - "single address instructions" (AC implicit operand)

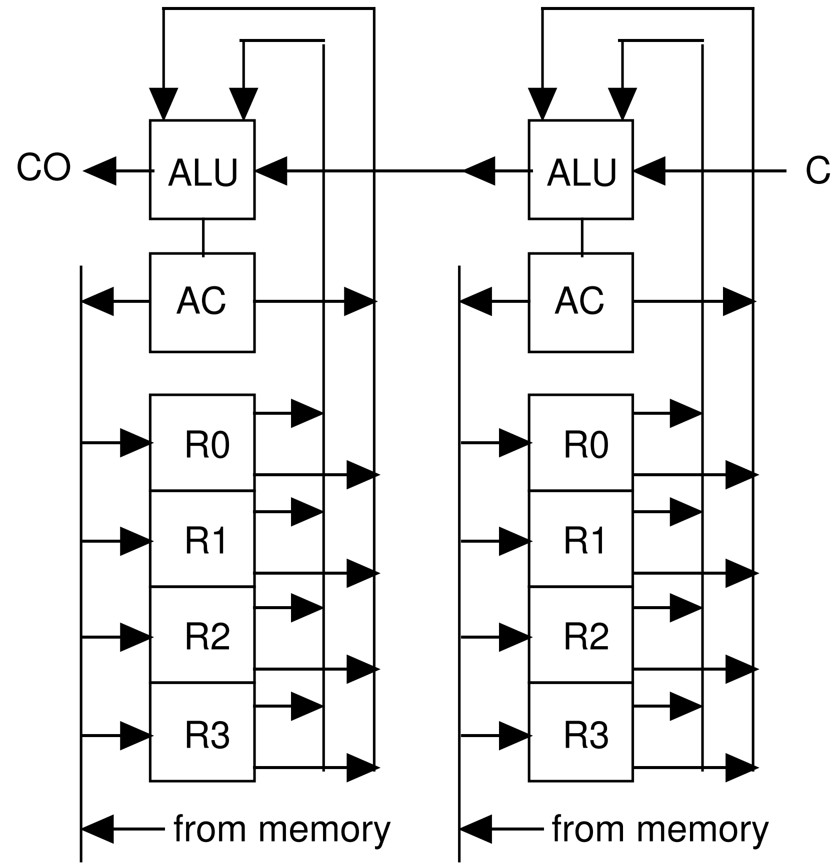


Data path (bit-slice)

- **Bit-slice concept – iterate to build n-bit wide datapaths**
  - may interate on two or four bit units as well (compatible with carry-lookahead)



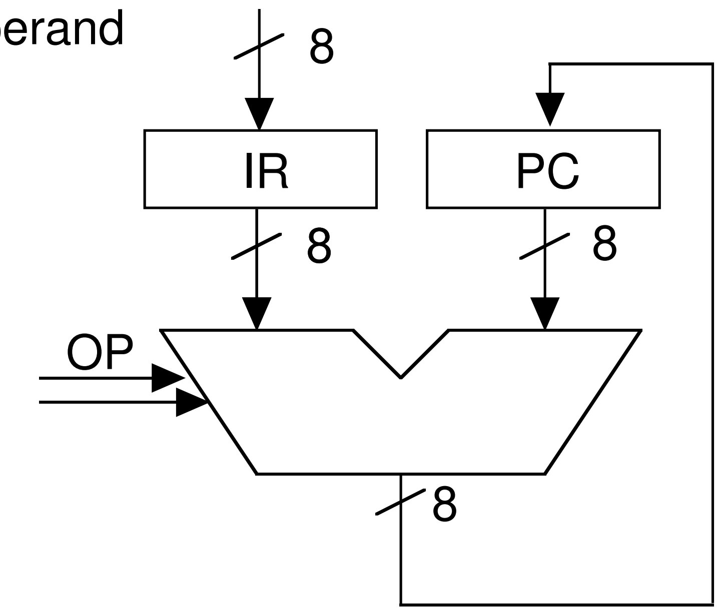
1 bit wide



2 bits wide

Instruction path

- **Program counter**
  - keeps track of program execution
  - address of next instruction to read from memory
  - may have auto-increment feature
- **Instruction register**
  - current instruction
  - includes ALU operation and address of operand
  - also holds target of jump instruction
- **Part of control**
  - may be incorporated into control FSM
- **Relationship to data path**
  - PC may be incremented through ALU



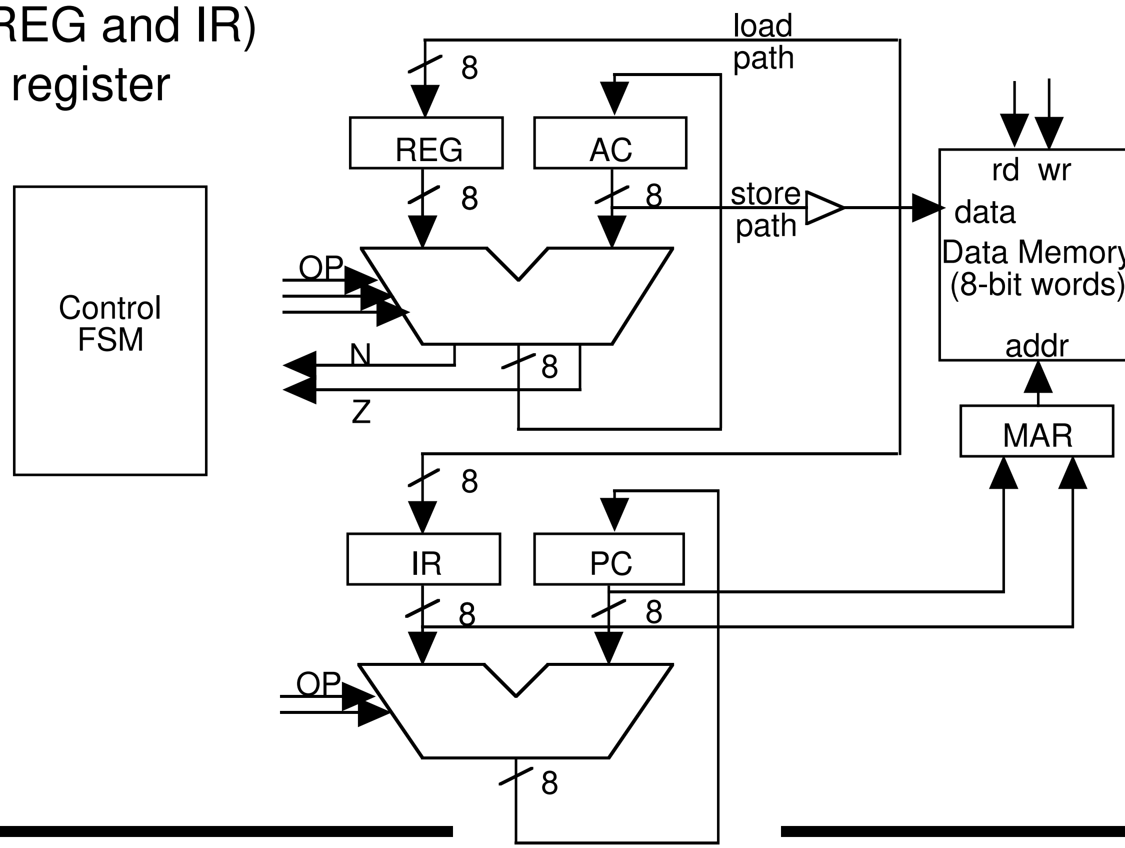
Data path (memory interface)

- **Memory**
  - separate data and instruction memory (Harvard architecture)
    - two address registers, two data registers
  - single combined memory (Princeton architecture)
    - single address register, single data register

- **Separate memory**
  - AC output goes to data memory input
  - REG input from data memory output
  - data memory address from instruction register
  - instruction memory address from program counter
  - IR input from instruction memory output

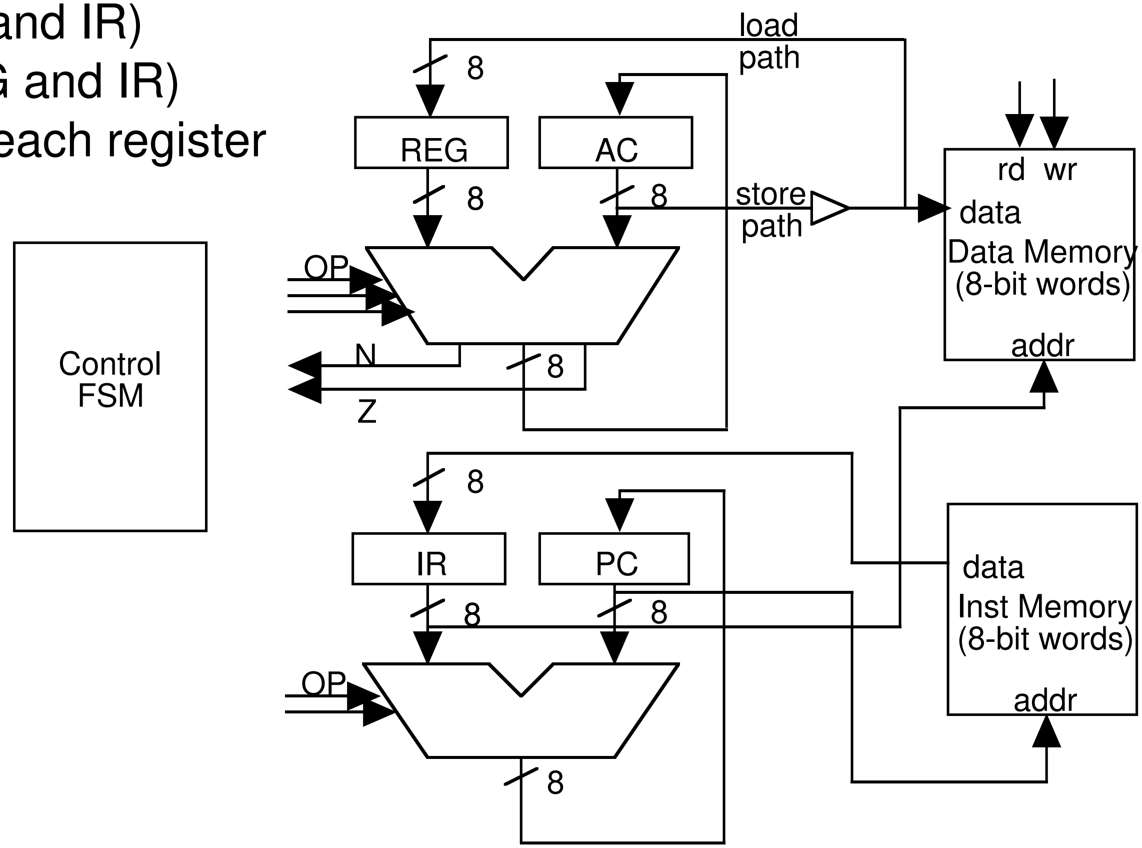
- **Single memory**
  - MAR (memory address register) loadable from PC or IR
  - MBR (memory buffer register) loadable from memory or AC

- **Register transfer view of Princeton architecture**
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register
  - MBR is split in two (REG and IR)
  - load control for each register



Block diagram of processor

- **Register transfer view of Harvard architecture**
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - two MARs (PC and IR)
  - two MBRs (REG and IR)
  - load control for each register



Tracing an instruction's execution

Instruction: **AC := AC + Mem<address>**

1. instruction fetch

move instruction address from PC to MAR
initiate a memory read sequence
move data from memory into register IR

2. instruction decode

op-code bits of IR are input to control FSM
rest of IR bits encode the operand address

3. operand fetch

move operand address from IR to MAR
initiate a memory read sequence
move data from memory into register REG

Tracing an instruction's execution

Instruction: AC := AC + Mem<address>

4. instruction execute

move data to ALU input
configure ALU to perform ADD operation
move result to AC

5. housekeeping

increment PC to point at next instruction

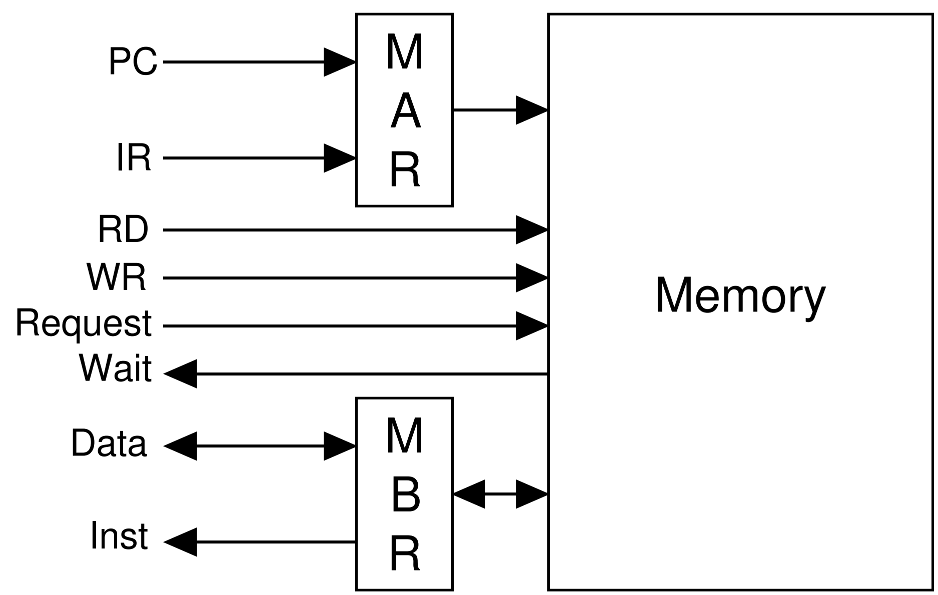
- **Control**
  - transfer data between registers by asserting appropriate control signals

- **Register transfer notation**

– instruction fetch:	PC $\rightarrow$ MAR;	– move PC to MAR
	Memory Read;	– assert memory READ signal
	Memory $\rightarrow$ IR;	– load IR from memory
– instruction decode:	IF IR<op code> = ADD_FROM_MEMORY THEN	
– instruction execution:	IR<addr> $\rightarrow$ MAR;	– move operand addr to MAR
	Memory Read;	– assert memory READ signal
	Memory $\rightarrow$ MBR;	– gate memory to MBR
	ALU ADD;	– instruct ALU to perform ADD
	ALU $\rightarrow$ AC;	– gate ALU result to AC
	PC+1 $\rightarrow$ PC;	– increment PC

- **How many states are needed to accomplish these transfers?**
  - data dependencies
  - resource conflicts

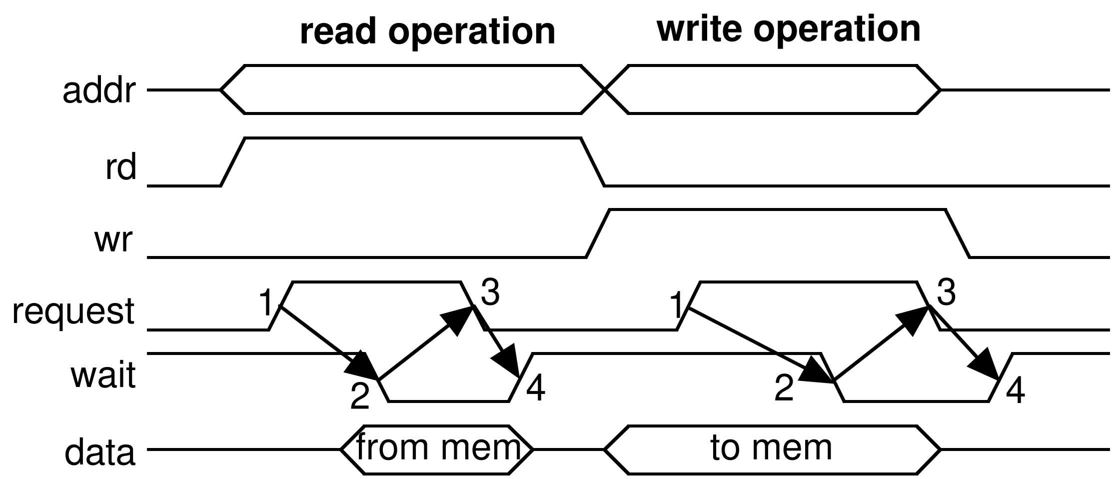
- **More realistic block diagram**
  - use MAR and MBR to decouple memory system from internal processor operation
  - don't have to necessarily be registers (could be muxes)
    - don't need to use a cycle to load
- **Issuing a memory request**
  - how long to wait for memory to complete operation
    - fixed number of cycles
    - memory may have signal that indicates when done
  - address stable
    - data, too, if write operation
  - assert RD or WR
  - assert Request
  - check until Wait goes low
  - four-cycle handshake



Memory interface (cont'd)

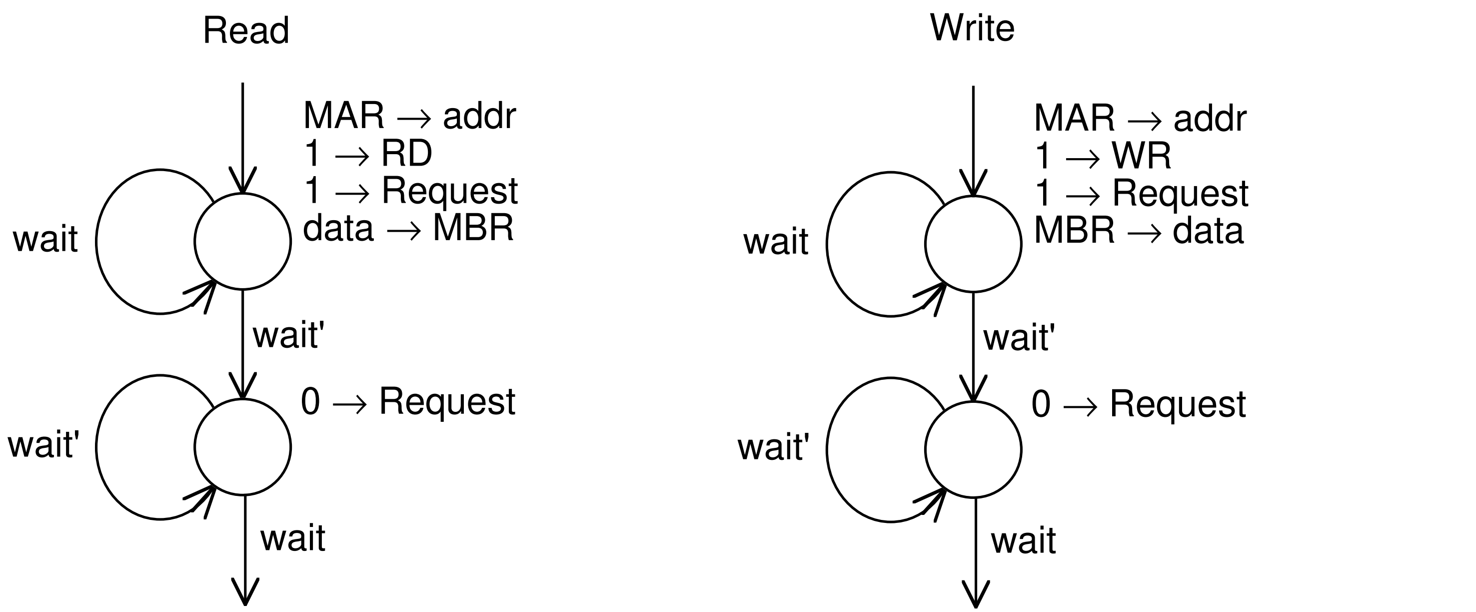
- **Handshaking**
  - follow 4-cycle handshake request/wait (ack) protocol
  - usually asynchronous (no common clock between CPU and memory)
  - may require use of synchronizer (adds delay to communication)

1. request asserted
2. wait unasserted
3. request unasserted
4. wait asserted



memory cannot make request unless WAIT signal is asserted
hi-to-lo transition on WAIT implies that data is ready (read)
or data has been latched by memory (write)

- **Memory interface – state diagram fragments for read/write cycles**



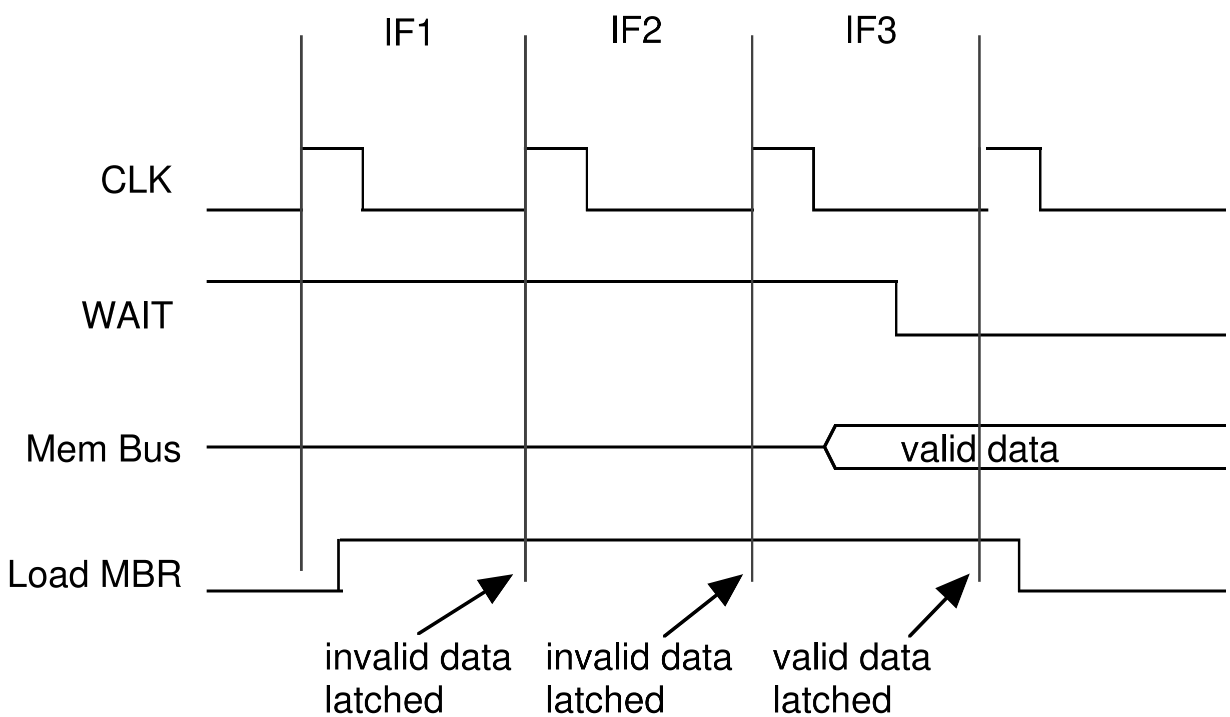
state 1: drive address bus
assert read request
catch data in MBR

state 2: unassert request
hold in state until WAIT reasserted
and handshake is completed

convention: if register transfer op is NOT asserted then it need not be mentioned in state diagram

Memory interface (cont'd)

- **Memory buffer register interface timing**
  - valid data latched on IF2 to IF3 transition
  - data must be valid before Wait can go low



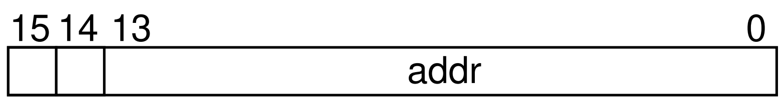
Input/output interface

- **Memory-mapped I/O**
  - I/O devices share the memory address space
  - control registers manipulated just like memory word
  - read/write register to initiate I/O operation
- **Polling**
  - program periodically checks whether I/O has completed
- **Interrupts**
  - device signals CPU when operation is complete
  - software must take over to handle the data transfers from the device
  - check for interrupt pending before fetching next instruction
  - save PC & jump to special memory location for interrupt handling subroutine
  - instruction set includes a "return from interrupt" instruction

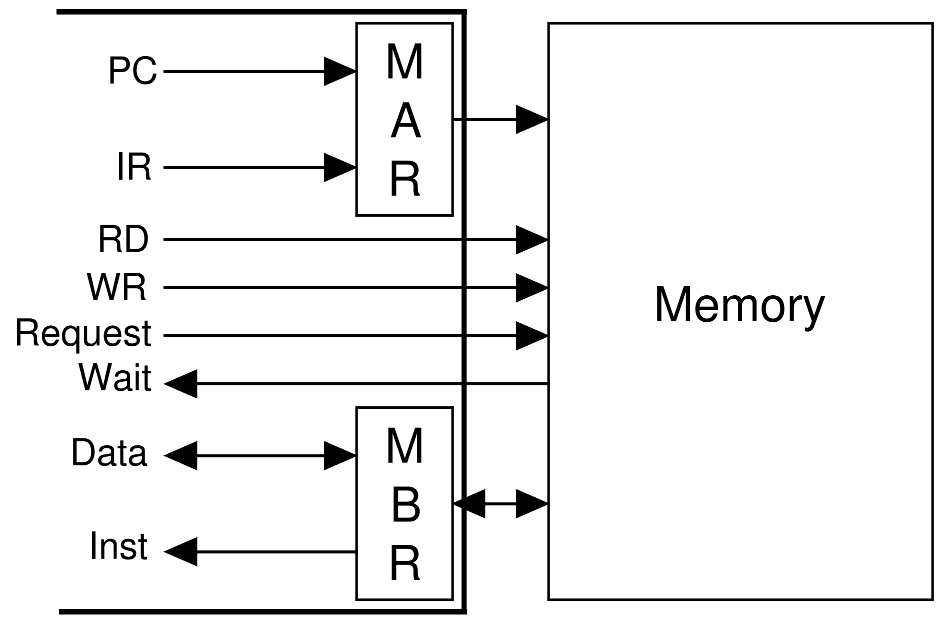
Instruction set

- **Instruction format**
  - load from memory: Mem[addr] $\rightarrow$ AC;
  - store to memory: AC $\rightarrow$ Mem[addr];
  - add from memory: AC + Mem[addr] $\rightarrow$ AC;
  - branch if accumulator is negative: if AC < 0 then addr $\rightarrow$ PC;

  - MAR 14 bits wide, MBR 16 bits wide

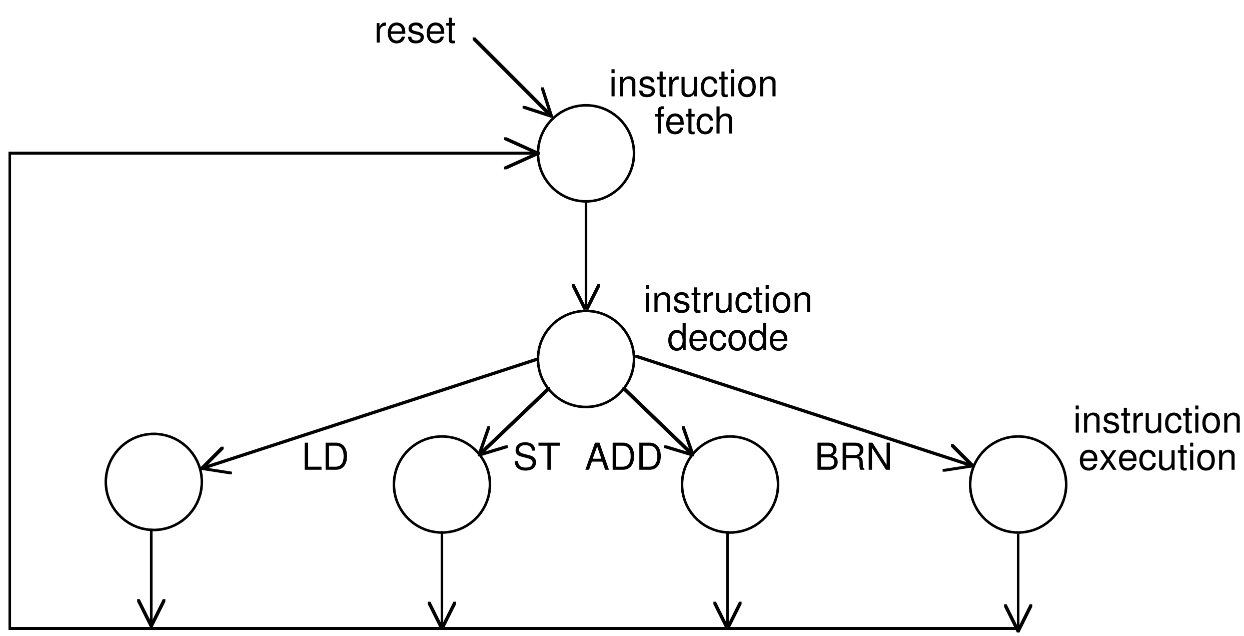


op code 00 = LD
01 = ST
10 = ADD
11 = BRN



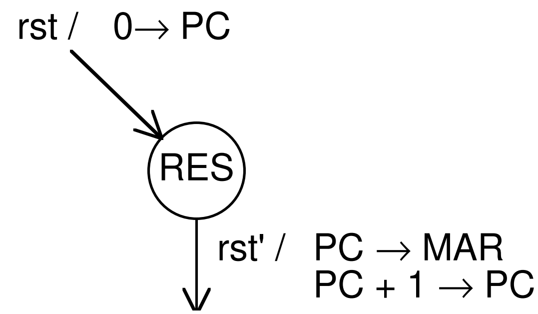
FSM controller for CPU (skeletal state diagram)

- **First pass at deriving the state diagram**
  - these will be further refined into sub-states



FSM controller for CPU (reset)

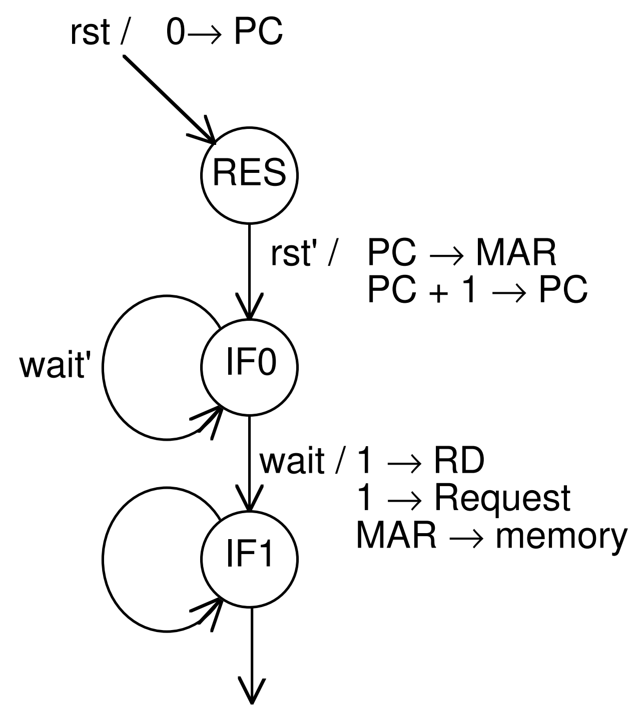
- **Assume synchronous Mealy machine**
  - outputs associated with arcs rather than states
- **Reset state and instruction fetch sequence**
- **On reset**
  - zero the PC
  - memory request unasserted
  - memory asserts WAIT signal



FSM controller for CPU (instruction fetch)

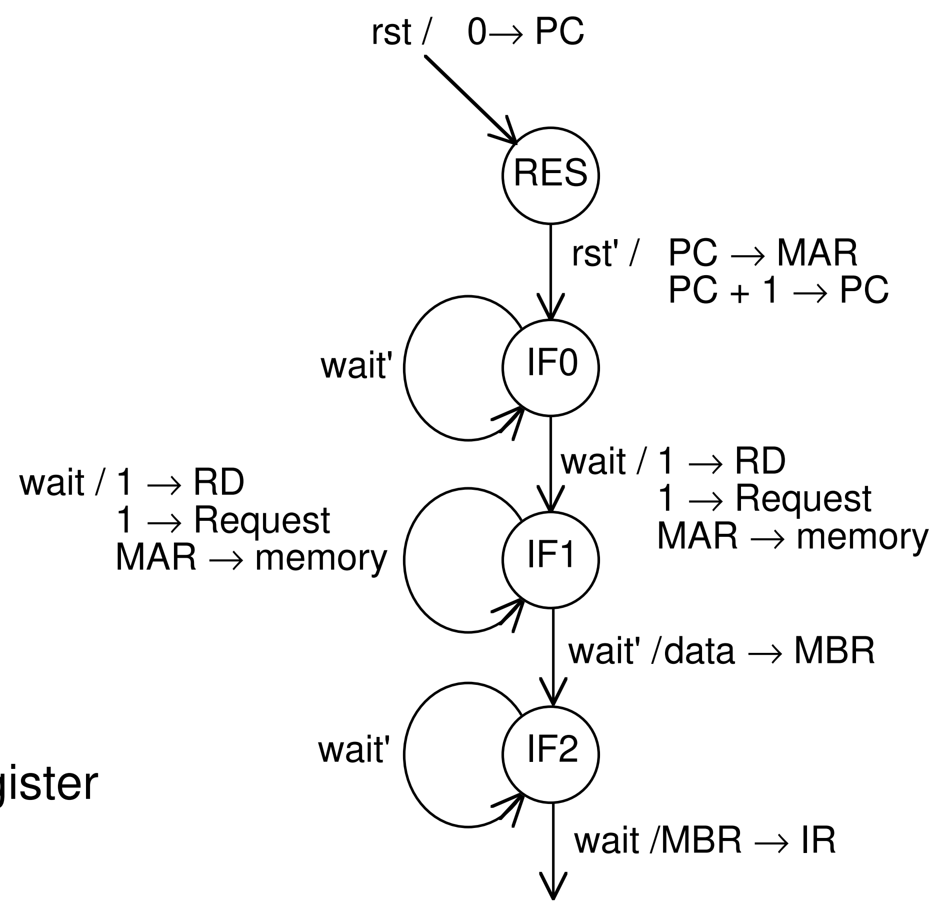
- **On reset**
  – zero the PC
  – memory request unasserted
  – memory asserts WAIT signal

- **Instruction fetch**
  – issue read request
  – 4-cycle handshake on WAIT signal



FSM controller for CPU (instruction fetch) (cont'd)

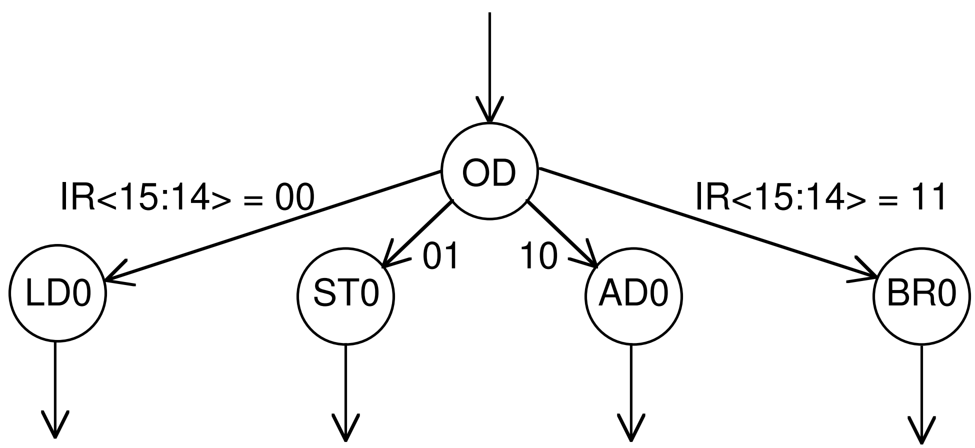
- **On reset**
  - zero the PC
  - memory request unasserted
  - memory asserts WAIT signal
- **Instruction fetch**
  - issue read request
  - 4-cycle handshake on WAIT signal



note: no explicit mention of how register transfers are implemented, however, register assumptions influence number of states

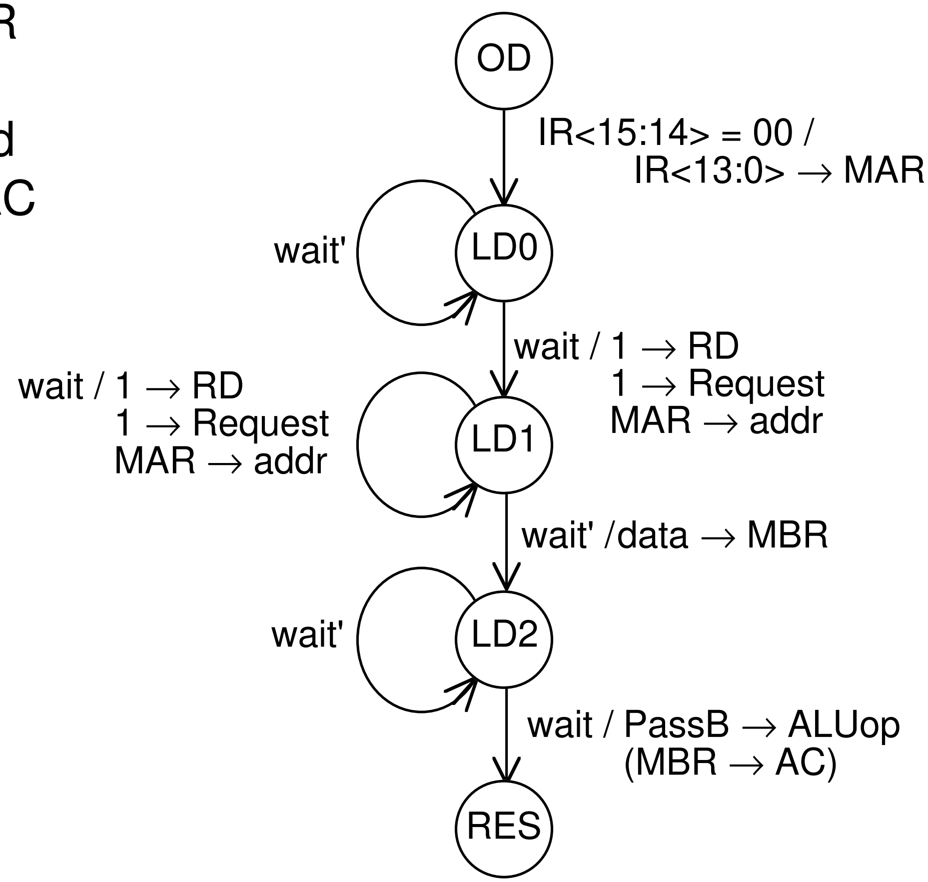
FSM controller for CPU (decode)

- **Operation decode state**
  - 4-way next state branch based on opcode bits
  - OD state may not be really needed
    - decode branching could be done on leaving previous state (part of instruction fetch)



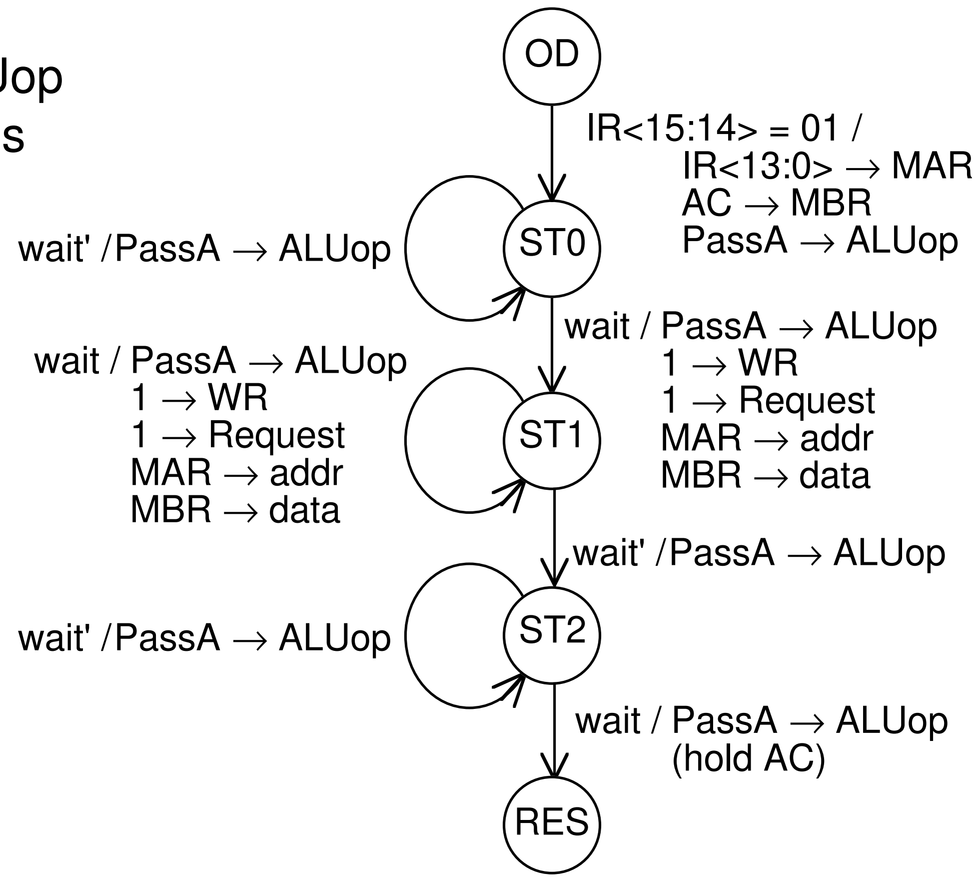
FSM controller for CPU (load instruction)

- **Load execution sequence – load AC from memory**
  - like instruction fetch, except that operand address comes from IR and data is loaded into AC
  - to load into AC need to first load into REG and then transfer to AC through ALU (path through functional unit)



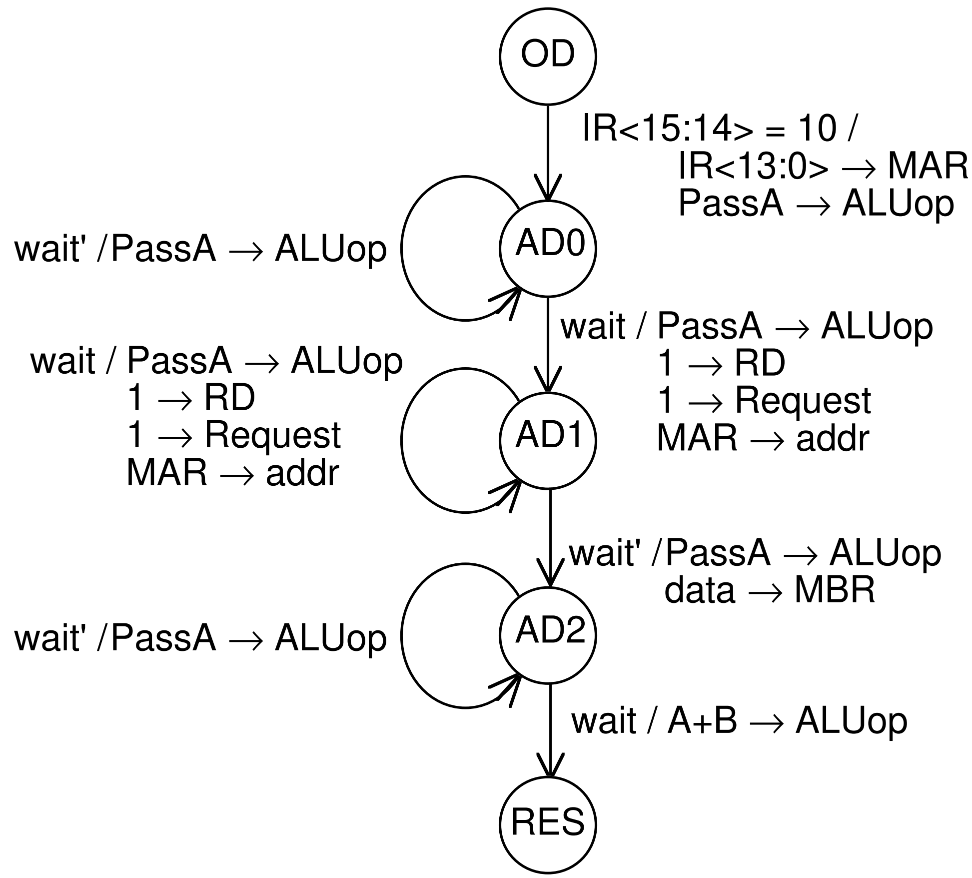
FSM controller for CPU (store instruction)

- **Store execution sequence – write value of AC to memory**
  - how do keep value in AC (it reloads every cycle): need to assert PassA on ALUop for every cycle when AC holds (go back and add this during instruction fetch as well)



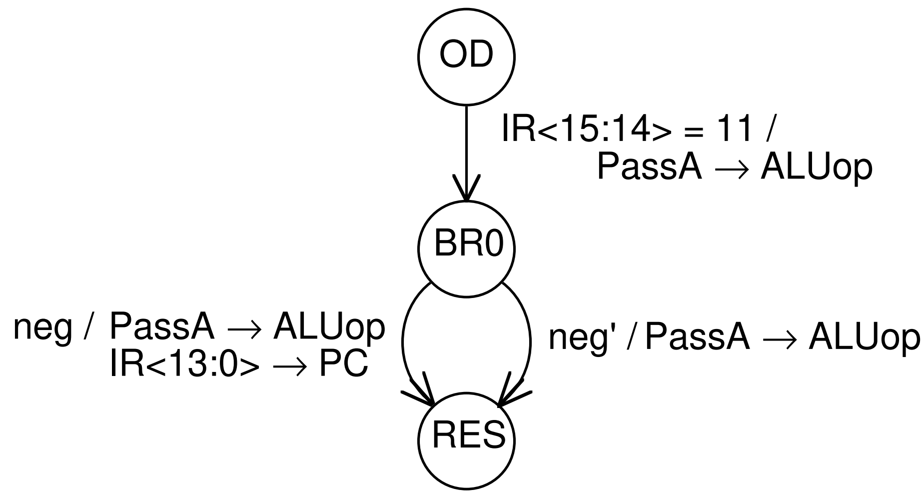
FSM controller for CPU (add instruction)

- **Add execution sequence – add value in memory to AC**
  - similar to load sequence
  - add MBR, AC rather than simply transfer MBR to AC



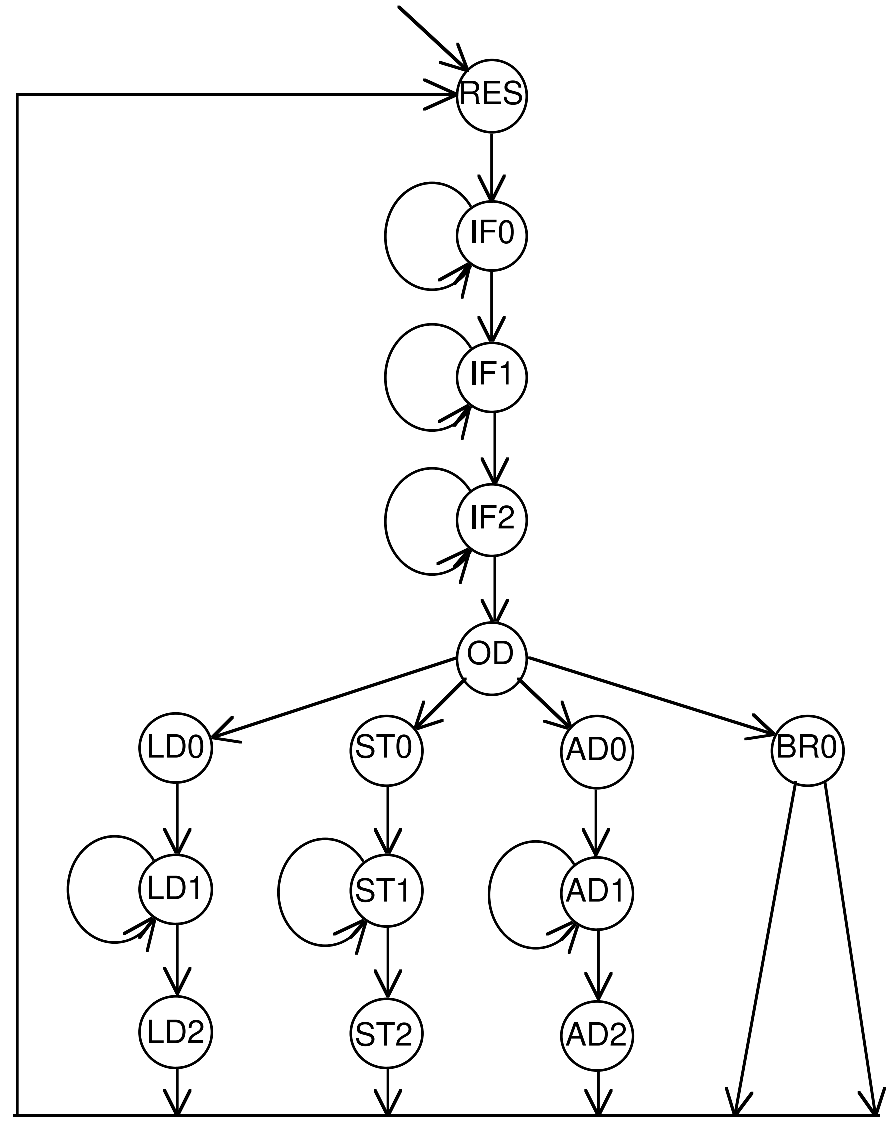
FSM controller for CPU (branch instruction)

- **Branch execution sequence**
  - replace PC with operand address if AC < 0, otherwise, do nothing



FSM controller for CPU (refined state diagram)

- **Refined state diagram**
  – simplify WAIT looping by eliminating some wait states (the fetch states do it once for all)

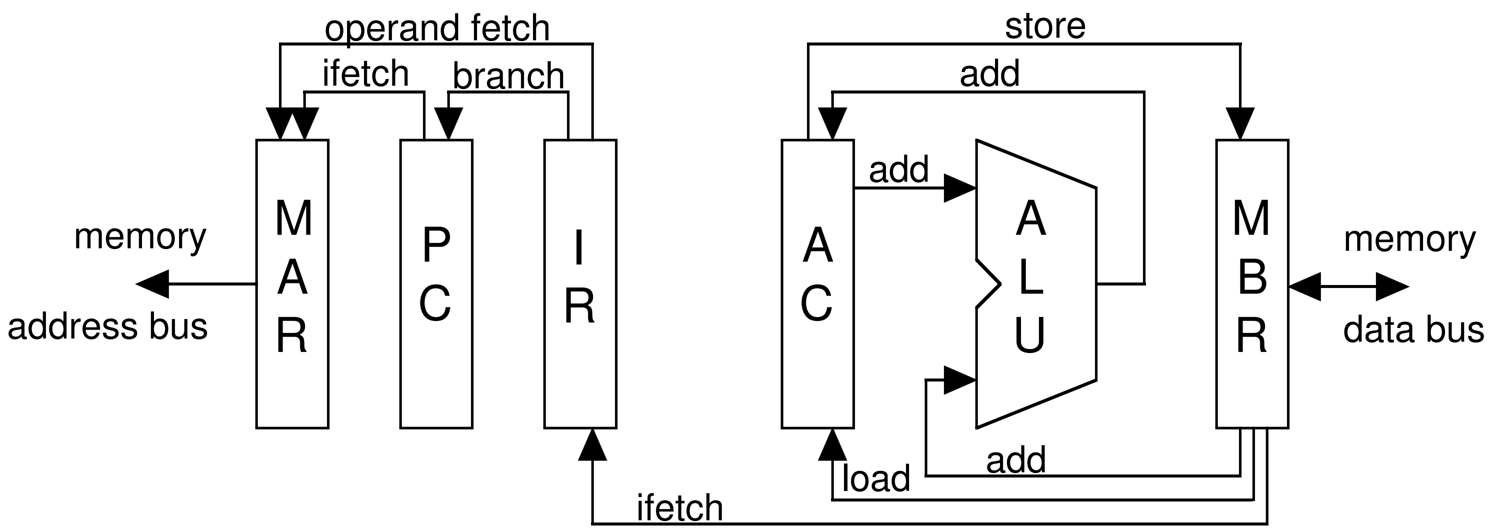


FSM controller for CPU (inputs and outputs)

- **State machines inputs and outputs**

Inputs	Outputs	
<u>from user</u>	<u>to memory</u>	
Reset	MAR → memory address bus	between memory and data path
	memory data bus → MBR	
<u>from memory</u>	MBR → memory data bus	
Wait	1 → RD	
	1 → WR	
<u>from data path</u>	1 → Request	
IR<15:14>		
Neg	<u>to data path</u>	
	0 → PC	
	PC + 1 → PC	
	PC → MAR	
	MBR → IR	
	MBR → AC	
	AC → MBR	
	AC + MBR → AC	
	IR<13:0> → MAR	
	IR<13:0> → PC	

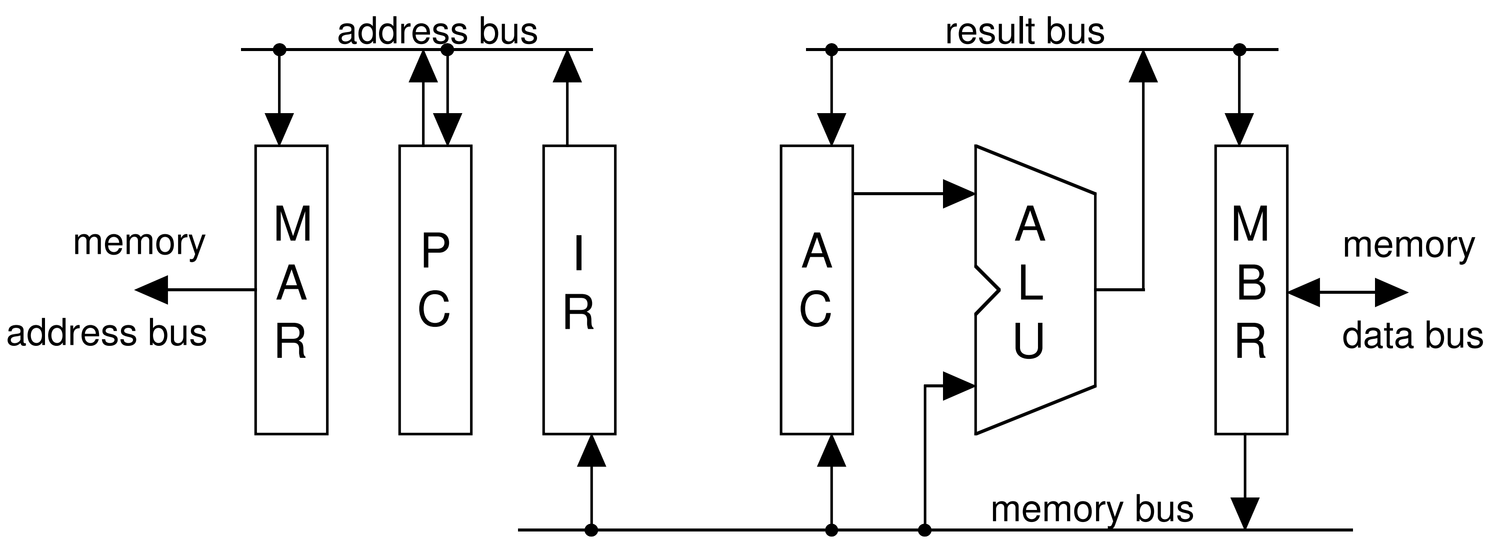
- **Specification has been mostly independent of data path interconnection**
  - lets map to a slightly different data path
  - this data path (like the previous one) uses a point-to-point interconnection scheme



Observations on data path interconnections

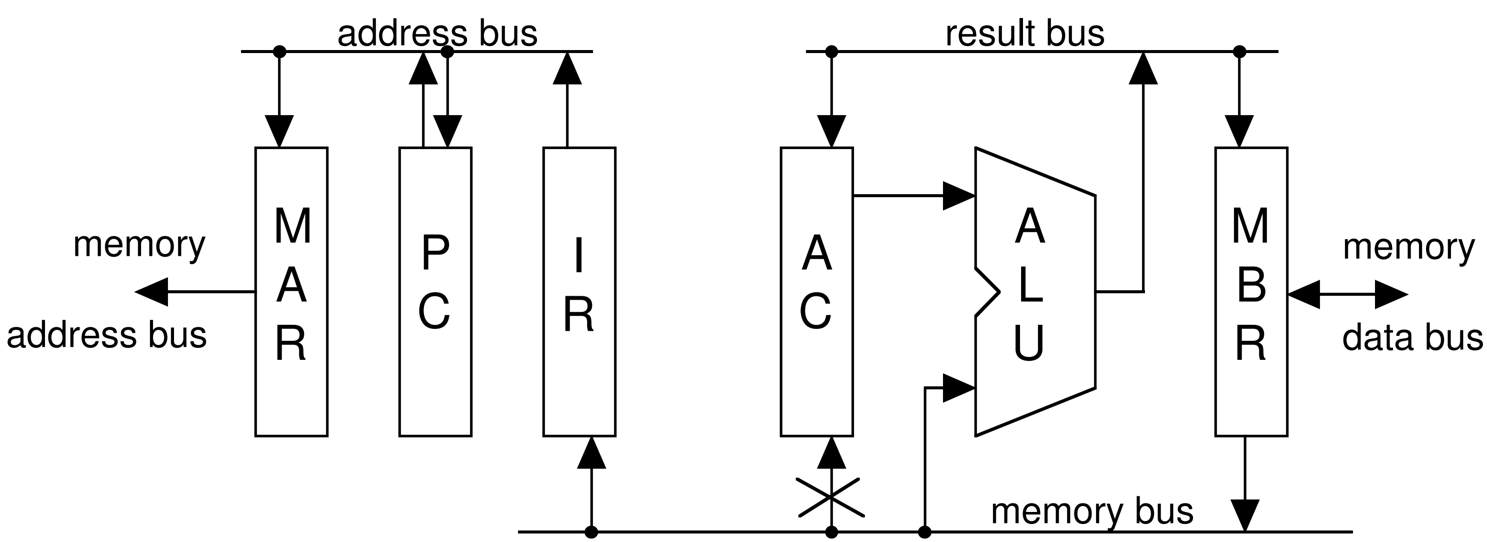
- **Instruction fetch and operand fetch take place at different times**
  - implies that IR/PC/MAR transfers can be implemented by single address bus
  - combine MBR, IR, ALU B, and AC connections on to memory bus
  - combine ALU, AC, and MBR connections on to result bus
  - AC $\rightarrow$ MBR implemeted through ALU using a "Pass A" operation

- **Three bus architecture**
  - AC + MBR $\rightarrow$ AC can be implemented in a single state



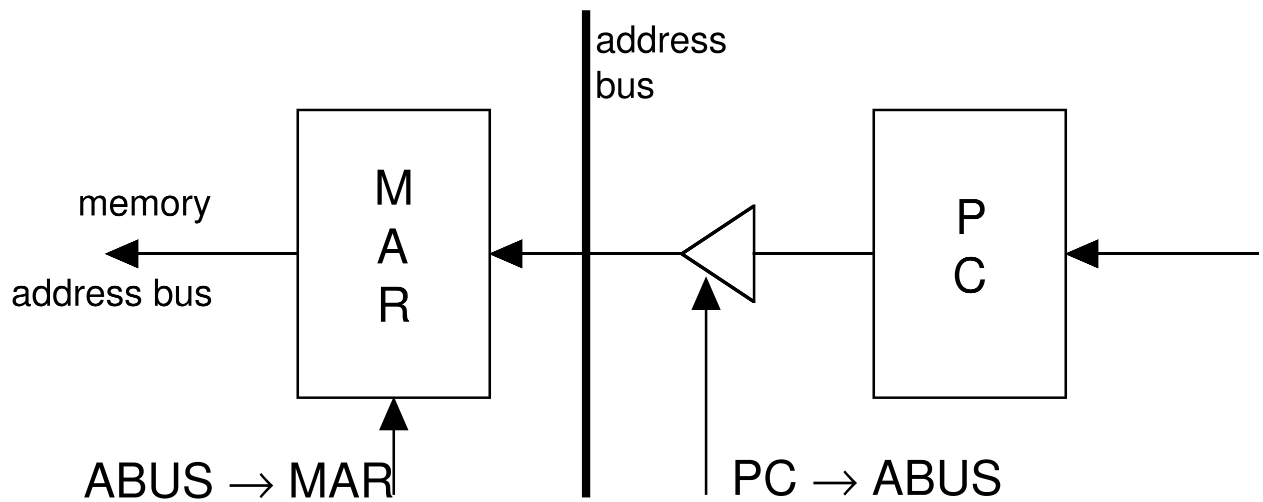
Observations on data path interconnections (cont'd)

- **AC has two inputs, RBUS and MBUS**
  - other registers except MBR have single input and output
  - dual ported configuration is more complex – requires multiplexer
- **Better idea: reuse existing paths were possible**
  - MBR $\rightarrow$ AC transfer can be implemented by "Pass B" ALU operation



Observations on data path interconnections (cont'd)

- **Detailed implementation of register transfer operations**
- **More detailed control operations are called microoperations**
  - one register transfer operation = several microoperations
  - some operations directly implemented by functional units:
    e.g., ADD, Pass B, 0 $\rightarrow$ PC, PC + 1 $\rightarrow$ PC
  - some operations require multiple control operations:
    e.g., PC $\rightarrow$ MAR implemented as PC $\rightarrow$ ABUS and ABUS $\rightarrow$ MAR



Micro-operations

- **Relationship between register transfer and microoperations**
  - delayed operations take effect at next clock cycle (register loads)

Register transfer	Micro-operations	
$0 \rightarrow PC$	$0 \rightarrow PC$	delayed
$PC + 1 \rightarrow PC$	$PC + 1 \rightarrow PC$	delayed
$PC \rightarrow MAR$	$PC \rightarrow ABUS$	immediate
	$ABUS \rightarrow MAR$	delayed
$MAR \rightarrow addr$	$MAR \rightarrow addr$	immediate
$data \rightarrow MBR$	$data \rightarrow MBR$	delayed
$MBR \rightarrow data$	$MBR \rightarrow data$	immediate
$MBR \rightarrow IR$	$MBR \rightarrow ABUS$	immediate
	$ABUS \rightarrow IR$	delayed
$MBR \rightarrow AC$	$MBR \rightarrow MBUS$	immediate
	$MBUS \rightarrow ALU\ B$	immediate
	ALU PASS B	immediate
	$ALU\ Result \rightarrow RBUS$	immediate
	$RBUS \rightarrow AC$	delayed

Micro-operations (cont'd)

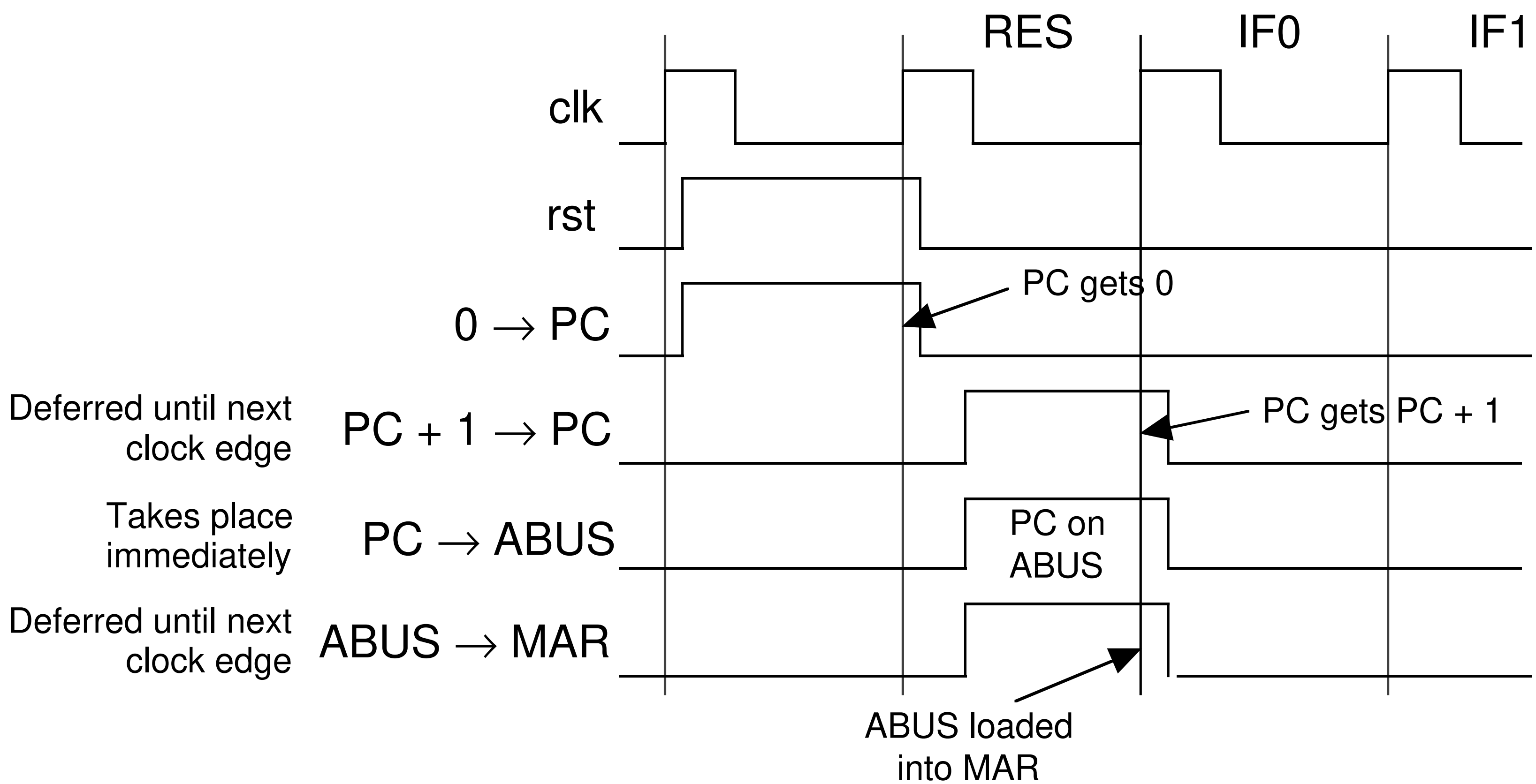
- **Relationship between register transfer and microoperations (cont'd)**

Register transfer	Micro-operations	
AC → MBR	AC → RBUS	immediate
	RBUS → MBR	delayed
AC + MBR → AC	AC → ALU A	immediate
	MBR → MBUS	immediate
	MBUS → ALU B	immediate
	ALU ADD	immediate
	ALU Result → RBUS	immediate
	RBUS → AC	delayed
IR<13:0> → MAR	IR → ABUS	immediate
	ABUS → IR	delayed
IR<13:0> → PC	IR → ABUS	immediate
	ABUS → PC	delayed
1 → RD	Read	immediate
1 → WR	Write	immediate
1 → Request	Request	immediate

Special microoperations for AC → ALU and ALU Result → RBUS not strictly necessary since these connections can be hardwired

Timing of micro-operations

- **Only register loads require waiting for clock edge (deferred or delayed)**
  - other operations can start immediately (immediate)



Section review

- **Computer design – an application of digital logic design procedures**
- **Datapath = functional units + registers**
  - ALU, registers, interconnection, busses
- **Control = finite state machine**
  - instruction interpretation = instruction fetch, decode, execute
  - relationship or register transfer operations to micro-operations
- **Memory interface**
  - MAR/MBR abstraction
- **Design of all basic CPU elements**